



InterActive 3 — Serial EEPROMs

Written By: Steven Robert Cypherd



PARTS:

- [Basic Stamp 2 \(1\)](#)

SUMMARY

I updated my program with full insert and delete. No networking or display. No plus formatting. Just a good E-Prom editor. Make sure you have strings before using insert or delete. It takes time to run insert and delete. I read and then write each string to its new place. My program does not check for limits on anything. The address is very important. If your address is off by a few bytes then you will lose those bytes from the front or back of your string. No errors happen. This $16 * tmp5 - 1$ looks OK. It is not OK. Basic Stamps are strictly left to right math. Basic Stamps 2s allow parentheses as $16 * (tmp5 - 1)$. With this $16 * tmp5 - 1$ is actually $(16 * tmp5) - 1$ and you would be off by 1 address. In this case I lost the first letter of the string I was editing.

Math can be fun. The Basic Stamp 2s support negative numbers, but you must be careful with them. If you have two variables counting down and you are comparing them $tmp3 >= tmp4$. If $tmp3$ reaches 0 and you subtract 1 it will be negative and more than $tmp4$. Put a debug statement on $tmp3$ and see where it goes. In this case my target is $tmp5$. I changed it to $tmp4 >= tmp5$ and let $tmp3$ go to 0. Then I adjusted $tmp5$ by -1 for the 0 based array of strings. Zeros are not good for math or comparing.

The main thing you need for an interactive device is memory. The cheapest and easiest

memory to use is a serial EEPROM. What is an EEPROM? It is an electrically programmable and electrically erasable memory with a serial interface. EEPROMs keep their memory contents when the power is shut off. EEPROMs let you re-program the memory, usually for ten million or so cycles. Consult the data sheet for your EEPROM for the exact specifications. You can store just about any type of data into an EEPROM. I wish I had friends to share this with. I will be working on LCD displays and kind of a dumb terminal to show how easy it is to connect your ideas to your people.

The program code zip is up on [Instructables](#) and [Let's Make Robots](#).

Most EEPROMs are byte-sized memory arrays. The biggest users of memory are strings like the quintessential "Hello World". Strings are an array of bytes in the processor's memory. Strings are well suited to EEPROMs. In this example my serial EEPROM is 2048/8 or 16K with 128 16-byte pages. Yes, 128 strings. Like most serial devices EEPROMs have a processor that is set up to write and read the memory array. You control them with commands and input and output leads. Timing is everything.

You need a lot of pins to use a serial EEPROM when you are writing to the memory. The pins: SO (Serial Output), SI (Serial Input), CK (Clock) and CS (Chip Select). I use separate pins for SO & SI to protect my EEPROM. You use commands to tell the EEPROM what you want it to do. To read a byte you lower CS and send in a Read command, such as `ShiftOut epRead\8, erAddr\16`. The `\8` tells ShiftOut to send 8 bits total. So 32 is 100000 in binary notation, which is only 6 bits. The `\8` makes it 00100000 and that is 8 bits. That is what the EEPROM needs. Reading is a continuous stream of bytes as long as you provide clock pulses. `ShiftIn myStr(tmp1)`. You can use an address with the read command. Each page is 16 Bytes. So 32 is page 1. Remember the EEPROM memory is zero-based and strings are zero-terminated.

Writing is a bit more complicated. You hold Write Protect and Hold pins high, then you set CS low and send a Write Enable command, such as `SHIFTOUT epSi, epCk, MSBFIRST, [epWrtEN\8]`. Then you raise CS, pause 5ms, lower CS and pause 5ms again. That enters the command into the EEPROM. Then you send in a Write command. `SHIFTOUT epWrite\8, erAddr\16`. When writing you usually need to write all 16 bytes in one command and terminate it correctly. In a loop I will use something like `SHIFTOUT myStr(tmp1)`. Then I pause 5ms and raise my CS to tell the EEPROM to write my data. You must wait or the EEPROM may crash. To reset the EEPROM you lower CS, wait 5ms, then raise CS, wait 5ms then lower CS and continue with your commands. I keep my CS line

low with a 10K resistor to ground.

The EEPROM also has a Write Protect pin. With this pin low you cannot write to the EEPROM. It also has a Hold pin; when it is low the EEPROM is off. You can write-protect one quarter, half or all of the memory with status flags. I keep Write Protect and Hold connected to V_{DD} .

Addressing is mandatory and easy. I use addresses on 16 Byte boundaries so it is easy. You can write any 16-byte page of memory that you need to and nothing else is affected. Truly random access memory on 128 16-byte strings. Remember the memory is zero-based and strings are zero-terminated. String 1 is actually string 0. All of my string numbering is one-based. My program takes care of the details. In my program `epAddr` is the string edit point and `myStrS` is how many strings I have. Displaying the strings starts at 0 and runs through `myStrS` numbering each string. With concatenated strings only the first string gets a number. Likewise, with sub-strings only the first one gets a number.

One thing about string handling is that most systems use zero-terminated strings. That means that the last byte in "Hello World0" is a zero byte. A 16 Byte array holds 15 characters and a terminating zero. Also most byte arrays are zero-based. That is, the first position is zero and the last is 15. For strings to be recognized you have to make sure that the zero is in the right place. You can use just one 16-byte array for everything. For short strings just move them to the front and terminate them. The rest of the string will be ignored. Keep track of your position. When you are concatenating a string to the next string you cannot use all 16 bytes. Why? The output routine still needs the first string terminated.

Other data needs to be formatted to fit into a `byte` array. A `word` has 16 bits or two bytes: a high byte and a low byte. `Longs` or `doubles` have 32 bits or four bytes: byte 3, byte 2, byte 1 and byte 0. You build formatters to take care of this type of data. Store them byte by byte in an order you like, like high byte then low byte. Load them back the same way. The memory of most processors is 16 bit pages. Word-sized memory that can be broken down into Bytes and on some down to the nybbles or bits for variables. When you store a `word` variable you usually have access to its bytes as variables too.

Some processors have good string handling functions. I am using a Parallax Basic Stamp 2 for this example. The commands used are `Debug`, `DebugIn`, `SerIn` and `SerOut`. `Debug` and `serOut` are the same function in most respects they just go to different places. To input a string you use a `STR` formatter that looks for character strings. `DebugIn STR`

`time\15\13` looks for and inputs a string up to 15 characters long that it loads into a byte array when the user presses Enter (13). If the string is less than 15 characters then the remaining characters are filled with zeros. The string “Hello” would be “Hello000000000000” in memory. `Debug STR time` prints “Hello”. The first zero terminates the string and the rest are ignored.

Numbers can be fun. The input formatters are for character strings. `DebugIn DEC time` looks at the string “hello123bye” and converts it to the number 123 and puts it into the variable `time`. It ignores the rest. It is useful for a lot of interactive things. For byte arrays you use control characters like zero as a terminator at the end of strings. You create other control characters to fit your needs. I use the ‘+’ to separate short strings and join long strings in my output routines. Short strings can be put together usually in the front of the EEPROM. Fifteen characters are never enough. You can create all sorts of controls in output routines.

My program: The button enters command mode. Enter a command and press Enter on the keyboard. You cannot keep these small processors waiting for an input. They are prone to crashing. You run an active loop waiting for a button to be pressed. My program stores the last address and the string count into the processor’s EEPROM and loads it at start up or re-boot. Set `myStrs` to zero for new strings and 1 for your strings to be loaded. My program is a fast string loader for an EEPROM.

My program uses the debug terminal and a standard button. You push the button to enter command mode. Type your input and press Enter. The LED should flash when waiting for a command. Remember to set the processor's EEPROM when you end a session.

